Efficient Dispatch of Multimethods in Constant Time Using Dispatch Trees

Eric Dujardin

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Efficient Dispatch of Multimethods in Constant Time Using Dispatch Trees

Éric Dujardin

N ° 2892

Mai 1996

THÈME 3

ISSN 0249-6399

Efficient Dispatch of Multimethods in Constant Time Using Dispatch Trees

Éric Dujardin

Thème 3 — Interaction homme-machine,
images, données, connaissances
Projet RODIN

Rapport de recherche n ° 2892 — Mai 1996 — 33 pages

Abstract: Optimizing method dispatch is a central issue in object-oriented language implementation. Besides overall efficiency, a major requirement for some languages and applications is constant-time performance. In systems with multi-methods, these requirements are still harder to meet. The compressed dispatch table scheme, presented in [AGS94], was the first to meet them. Its compression algorithm is based on the definition of *pole types*. In this report, we investigate another structure, namely the dispatch tree, showing that it can also meet those requirements. We show that pole types can be used to compress dispatch trees, and we describe some optimizations of compressed dispatch trees. The resulting run-time structures are yet smaller than compressed dispatch tables.

Key-words: multi-methods, run-time dispatch, pole types.

(Résumé : tsvp)

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : (33 1) 39 63 55 11 – Télécopie : (33 1) 39 63 53 30

Sélection efficace des multi-méthodes en temps constant par des arbres d'envoi

Résumé : L'optimisation de l'envoi de méthode est un aspect essentiel de la mise en œuvre des langages orientés-objet. En plus de l'efficacité globale, certains langages et certaines applications demandent une sélection en temps constant. Dans les systèmes à multi-méthodes, il est encore plus difficile de repondre à ce besoin. Le principe des tables d'envoi comprimées, présenté par [AGS94], a été le premier à y répondre. Son algorithme de compression se base sur la définition de *types pôles*. Dans ce rapport, nous étudions une autre structure, l'arbre d'envoi, et montrons qu'il peut également y répondre. Nous montrons que les types pôles peuvent être utilisés pour comprimer les arbres d'envoi, et nous décrivons quelques optimisations des arbres d'envoi comprimés. Les structures d'exécution résultantes sont notablement plus petites que les arbres d'envoi comprimés.

Mots-clé : Multi-méthodes, sélection dynamique, types pôles.

1 Introduction

The efficiency of method dispatch is a central issue in object-oriented languages. Many proposals aim at improving it, providing efficient method dispatch either in variable or in constant time. Variable-time solutions are generally based on caches (as in [UP83, KR90, DS84, CU91]), which offer a good overall performance, and code analysis and rewriting techniques (as in [GR89, UP83, SCB+86, Mey92, CU91]) allow to reduce the number of methods that may be selected for a given invocation, possibly resolving it at compile-time. Constant-time techniques use a precalculated two-dimensional table, where e.g. types appear as column indices and method names as line indices. Run-time dispatch then comes down to an access in this table. As this leads to tables too large in practice, optimization techniques reduce them by removing their empty cells, as in [DMPM89, ES92, HC92, DH95].

Efficient, constant-time run-time dispatch techniques allow to predict the execution time of run-time dispatch, which is necessary in some contexts like real-time systems or query optimization. Moreover, these techniques are also complementary of variable-time techniques, to be used e.g. when code analysis cannot statically resolve an invocation, or in case of cache miss.

In multi-method systems, method dispatch depends not only on the run-time type of one expression, as it is the case with ordinary methods, but of several expressions. The number of possible type combinations is therefore higher, which should cut down the efficiency of variable-time solutions. Indeed, it not so likely that the types of several expressions be sufficiently known at compile-time, to statically resolve an invocation. Cache misses should also occur more often, because of the number of type combinations. When these solutions fail, the performance penalty is also higher if the implementation then relies on a naive dispatch technique, because method dispatch is more complex. Therefore, a constant-time solution, efficient for all invocations, is more needed.

An approach that meets the requirements of an efficient and constant-time dispatch for multi-methods is presented in [AGS94]. It is based on the simple principle of dispatch tables, in which each argument position of an invocation is associated with a dimension of the table. Therefore method dispatch comes down to an access in a n-dimensional table. These tables

must be compressed to be practically usable. This is achieved by defining them only for some types, called pole types. This compression technique allows both to remove empty $n-1$-dimensional planes of the table, and to group identical $n-1$-dimensional planes.

In this report, we investigate another dispatch structure, namely the dispatch tree. Trees are more complex than tables but they can also be compressed using pole types. Compressed trees have the potential of using less memory than compressed tables, because the removal and grouping of cells applies at a finer grain. In this report, we show that compressed trees allow efficient and constant-time dynamic dispatch, and we compare the sizes of compressed tables and trees on a real application with multi-methods.

The report is organized as follows. Section 2 introduces preliminary notions and notations on multi-methods and dispatch. Section 3 gives an intuitive description of poles, of their use in dispatch table compression, and of dispatch trees. Section 4 formally defines compressed dispatch trees. Section 5 describes two size optimizations of dispatch trees. Section 6 discusses their implementation and gives some experimental results. Finally, section 7 concludes and compares our results with previous work on dispatch trees.

2 Background

We briefly review some terminology mainly introduced in [ADL91]. We denote *subtyping* by $\preceq$. Given two types T_1 and T_2, if $T_1 \preceq T_2$, we say that T_1 is a subtype of T_2 and T_2 is a supertype of T_1. The set of types is denoted Θ. A generic function is defined by its name and its arity (in Smalltalk parlance, a generic function is called a *selector*). To each generic function m of arity n corresponds a set of methods $m_k(T_k^1, \ldots, T_k^n) \rightarrow R_k$, where T_k^i is the type of the ith formal argument, and where R_k is the type of the result. We call the list of arguments $(T_k^1, \ldots, T_k^n)$ of method m_k the *signature* of m_k. We also define a relation $\preceq$ on signatures, called *precedence*, such that $(T^1, \ldots, T^n) \preceq (T'^1, \ldots, T'^n)$ iff for all i, $T^i \preceq T'^i$. An invocation of a generic function m is denoted $m(T^1, ..., T^n)$, where $(T^1, \ldots, T^n)$ is the signature of the invocation, and the T^i's represent the types of the expressions passed as arguments.

In traditional object-oriented systems, generic functions have a single specially designated argument – called *receiver* or *target* – whose run-time type is used to select the most applicable method to execute. Such methods are called *mono-methods*. *Multi-methods*, first introduced in CommonLoops [BKK+86] and CLOS [BDG+88], generalize mono-methods by considering that all arguments are targets. Multi-methods are now a key feature of several systems such as Polyglot [ADL91], Kea [MHH91], Cecil [Cha92], and Dylan [App94]. They have been integrated as part of the SQL3 proposition [Mel94] for a standardized database query language .

Object-oriented languages support *late binding* of method code to invocations: the method that gets executed is selected based on the run-time type(s) of the target argument(s). This selection process is called *method dispatch*. Operationally, method dispatch looks for the method whose signature most closely matches the run-time type(s) of the target argument(s). This method is called the *Most Specific Applicable* (MSA) method. In the rest of this report, we assume that for any function invocation, if there is an applicable method, then there always exists a unique MSA method.

The basis of method specificity is a precedence relationship called *argument subtype precedence* in [ADL91] : a method m_i is more specific than a method m_j if all the target arguments of m_i are subtypes of the target arguments of m_j. However, such a relationship does not yield a total order in the presence of multiple inheritance or multi-targeting and cannot guarantee the UMSA property. Thus, additional ordering criteria can be used to obtain a total order.

Monotonicity is defined in [DHHM94] as the intuitive property that if a method m_1, applicable to an invocation $m(s)$, is not chosen as its MSA method, then m_1 cannot be the MSA method of invocations whose signatures is more specific than s. Although languages differ in the way they complement argument subtype precedence ordering, our results are applicable to every language that enforces at least argument subtype precedence with monotonicity.

3 General Principle

The main idea of the compression technique is to reduce the set of invocations to which dispatch structures directly apply, and to map all possible invocations to this reduced set. Instead of storing in the dispatch structures the MSA method of each possible invocation signature, the algorithms store in these structures the MSA of signatures composed of a reduced set of types, called *pole types*, or *poles*. In this section, we first give an intuitive description of poles and dispatch tables, then we explain why dispatch trees can be built using poles to provide a better compression rate.

3.1 Pole Types and Dispatch Tables

Consider the example of types and methods from [AGS94] in Figure 1. On the first argument position of m, types A and C behave in the same way. Indeed, for all invocation signatures which first argument type is C, the result of dynamic dispatch is the same if we substitute C with A as first argument type. The same substitution would also apply to a F as first argument. We say that C and F are in the *1-influence* of A, or that A is the 1-pole of C and F. Similarly, G and F are in D's 1-influence, and E and I are in B's 1-influence. The result of dynamic dispatch for invocations in which D is the first argument's type depends on the precedence ordering, but the same result is obtained if D is substituted with G or H. Again, G and H are in D's influence.

As a consequence, we can reduce dispatch structures to signatures made of poles, or pole signatures. The simplest structure to hold the MSA method of all invocation signatures is the table. With each argument position of an n-ary generic function, is associated a dimension in the dispatch table. In a compressed dispatch table, only the i-poles of the generic function are row indices of dimension i. Supplementary tables, called argument-arrays, relate all types to their poles.

Consider the tables in Figure 2. The table on the right is the compressed dispatch table, that holds the method numbers for each pole signature of method m, as shown in Figure 1. These method numbers are found using argument subtype precedence and the complementary precedence orders $m_1 < m_3$ and $m_2 < m_5$. As m has two target arguments,

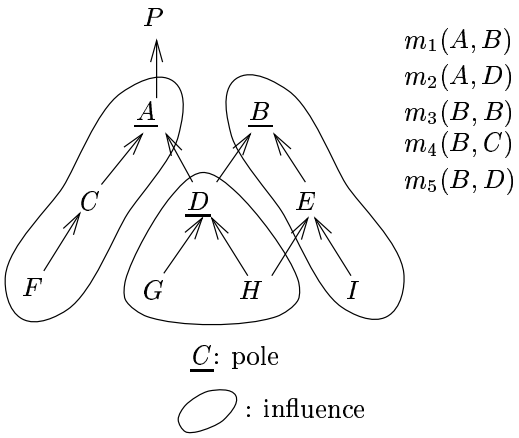


Figure 1: Example Type Graph with 1-Poles and 1-Influences

	P	A	B	C	D	E	F	G	H	I
m_arg1	0	A	B	A	D	B	A	D	D	B
m_arg2	0	0	B	C	D	B	C	D	D	B

	B	C	D
A	1	–	2
B	3	4	5
D	1	4	2

Figure 2: Argument-Arrays and Compressed Dispatch Table

this table is bi-dimensional. The 1-poles A, B and D label the lines, and the 2-poles B, C and D label the columns. The table on the left holds together the two argument-arrays m_arg1 and m_arg2. Now consider the invocation $m(C, D)$. At run-time, m_arg1 is used to find that the 1-pole of C is A, hence the method to select appears in the "A" line of the dispatch table. Likewise, m_arg2 is used to find that the 2-pole of D is D, hence the method to select appears in the "D" column of the dispatch table. Indeed, the MSA method of $m(C, D)$ is m_2.

3.2 Dispatch Trees

The dispatch tree of a n-ary generic function m is a directed, balanced tree of depth n. Each invocation signature is associated with a unique path in this tree. This path starts from the root of the tree, and each type of the signature determines the choice of one of the successive

branches of this path. The vertices that are leaves of this tree, are labelled with the MSA methods of the invocations.

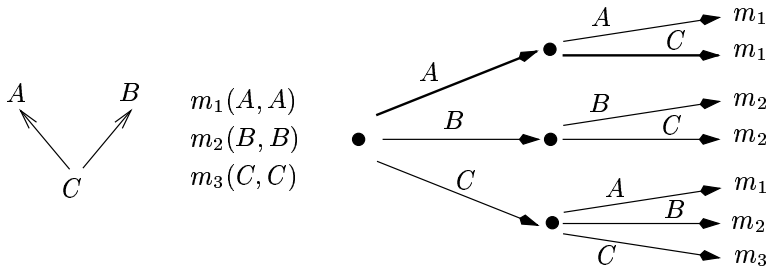


Figure 3: Type Graph and Generic Function With Dispatch Tree

Example 3.1: Consider in Figure 3 the graph of types A, B and C, the generic function m and the dispatch tree associated with m. The branches of this tree are labelled with the types that determine the choice of these branches at run-time. Each well-typed signature is associated with a path in the dispatch tree, that leads to a leaf labelled with the corresponding MSA method. For signature (A, C), the path starting from the root and made of the branches labelled with first A, then C (these branches appear in bold in Figure 3), leads to m_1 which is the MSA method of $m(A, C)$. Concerning (A, B), there is no associated path as $m(A, B)$ has no MSA. □

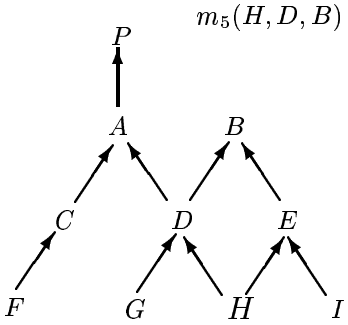
In an uncompressed dispatch tree, each vertex of depth i is associated with only one beginning of signature $(T^1, \ldots, T^{i-1})$, and from this vertex starts one branch for each valid type T^i of Θ. T^i is valid if there exists an invocation $m(T^1, \ldots, T^{i-1}, T^i, \ldots, T^n)$ which has an MSA method. For 100 types, this yields a tree with $1 + 100 + 100^2 + \ldots + 100^(n-1)$ vertices.

As in the case of dispatch tables, poles allow to compress dispatch trees. Indeed, as the invocation signatures have the same MSA method as their pole signatures, it suffices to build the trees only for pole signatures. This comes down to associate the branches with poles only. This alone does not bring a better compression than that of dispatch tables, but dispatch trees can be further compressed. The idea is the following: for a vertex of depth i (with $1 < i < n$), reached from a beginning of pole signatures $(T_p^1, \ldots, T_p^{i-1})$, only some

methods are applicable to this beginning of signature. The subtree that starts from this vertex allows to perform the selection between these methods, which is done by taking into account the next types of the invocation signature. To build the branches of this subtree, it is only necessary to consider the poles associated with this method subset, instead of the poles associated with all the methods of the generic function.

Example 3.2: In the preceeding tree, all types are poles. Consider however the vertices reached using a beginning of signature (A) or (B): the methods applicables to them are respectively m_1 and m_2, that define as respective sets of 2-poles $\{A\}$ and $\{B\}$. Hence the tree can be compressed, simply by suppressing the branches starting from these vertices, and labelled with C, because they lead to the same MSA methods as the other branches starting from the same vertice. □

This principle applying to every depth i, a progressive decrease in the number of poles is obtained while going to the leaves. Poles used in a vertex of depth i, to define the branches leading to the following vertices, are called the *local poles* of this vertex. When n is large enough, this principle allows to build dispatch trees actually smaller than dispatch tables. In practice, this generally happens for $n \geq 3$.



	A	B	D
	Plane A		
C	1	3	3
B	–	–	–
D	4	–	4
	Plane E		
C	–	–	–
B	2	–	2
D	2	–	2
	Plane H		
C	1	3	3
B	–	–	–
D	4	5	4

Figure 4: 3-Dimensional Dispatch Table

Example 3.3: Consider the 3-ary generic function m in Figure 4. Its compressed dispatch table is a cube presented as 3 planes. Planes are labelled with 1-poles, lines are labelled with 2-poles, ans columns are labelled with 3-poles. The planes of this table could separately be further compressed. For example, in plane E the lines 2 and 3 are identical and line 1 is empty. These compressions cannot apply to the dispatch table because they are local to some planes. On the contrary, a tree structure allows them, as shown in Figure 5. In this tree, there is only one path associated with E as first argument. As we shall see, this tree takes fewer memory than the corresponding dispatch table. □

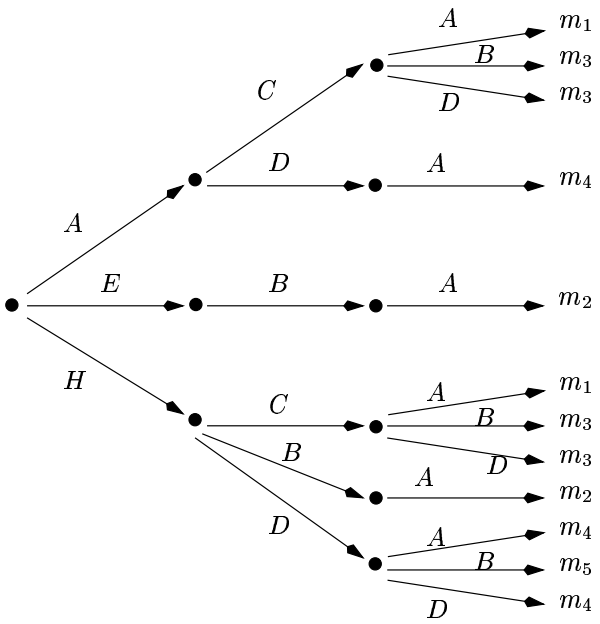


Figure 5: Dispatch Tree

4 Dispatch Tree

Before presenting the formal definition of a dispatch tree, we first give some definitions relative to multi-methods.

4.1 Selection of Multi-Methods

For the rest of this report, we call Θ the set of existing types, and we consider a generic function m of arity n, such that the signature of method m_k is $(T_k^1, \ldots, T_k^n)$. M is the set of all methods m_k. We first review some notations and results introduced in [AGS94].

Cover. The cover of a type T, noted $cover(T)$, is the set of subtypes of T:

$$cover(T) = \{T' \mid T' \preceq T\}$$

Example 4.1: In the schema of Figure 1, we have $cover(A) = \{A, C, D, F, G, H\}$. □

The cover of a set of types $\{T_1, \ldots, T_n\}$, noted $cover(\{T_1, \ldots, T_n\})$, is the union of the covers of each type of the set:

$$cover(\{T_1, \ldots, T_n\}) = \bigcup_{i=1}^{n} cover(T_i)$$

Ith Static Arguments. The ith static arguments of a generic function m, noted $Static_m^i$, are the types of the ith formal arguments of the methods of m:

$$Static_m^i = \{T \mid \exists m_k \text{ with } T_k^i = T\}$$

Example 4.2: In the schema of Figure 1, we have $Static_m^1 = \{A, B\}$, and $Static_m^2 = \{B, C, D\}$. □

Ith Dynamic Arguments. The ith dynamic arguments of a generic function m, noted $Dynamic_m^i$, are the cover of the ith static arguments of m:

$$Dynamic_m^i = cover(Static_m^i)$$

They represent the types that can appear at the ith position in invocations of m at run-time. $Dynamic_m$ is the cross product of the ith dynamic arguments sets:

$$Dynamic_m = \prod_{i=1}^{n} Dynamic_m^i$$

Example 4.3: In the schema of Figure 1, we have $Dynamic^1_m = \{A, C, D, F, G, H\}$, and $Dynamic^2_m = \{B, C, D, E, F, G, H, I\}$. □

Well-Typed Signature. An invocation signature $m(T^1, \ldots, T^n)$ is well-typed if and only if there is an MSA method for this signature.

I-Pole. A type $T \in \Theta$ is an i-pole of generic function m, $i \in \{1, \ldots, n\}$, denoted $is\_pole^i_m(T)$, iff:

$$T \in Static^i_m \text{ or } |min_{\preceq} \{T' \in \Theta \mid is\_pole^i_m(T') \text{ and } T' \succ T\}| > 1$$

The set of i-poles of m is denoted $Pole^i_m = \{T \mid is\_pole^i_m(T) = \text{true}\}$. We call *closest-poles*$^i_m(T)$ the set $min_{\preceq}\{T_p \in Pole^i_m \mid T \prec T_p\}$. The poles that are included in $Static^i_m$ are called *primary poles*, and the others are called *secondary poles*.

Example 4.4: We determine the 1- and 2-poles of the schema of Figure 1. As $Static^1_m = \{A, B\}$, A and B are 1-poles. Moreover, they are the closest poles of D, so D is also a 1-pole. So $Pole^1_m = \{A, B, D\}$. As $Static^2_m = \{B, C, D\}$, B, C and D are 2-poles. There is no secondary pole, so $Pole^2_m = \{B, C, D\}$. The 1-poles are underlined on Figure 1. □

I-Influence. Every i-pole T_p of a generic function m is associated with its influence, noted $influence^i_m(T_p)$ and defined as:

$$Influence^i_m(T_p) = \{T \preceq T_p \mid \forall T'_p \in Pole^i_m, T \not\preceq T'_p \text{ or } T_p \preceq T'_p\}.$$

Example 4.5: We determine the 1-influence of our 1-poles. The 1-influence of A contains A, C and F as A is their only 1-pole supertype. For the same reason, the 1-influence of B contains B, E and I. D does not belong to the 1-influence of A or B as D is a 1-pole, and as pole it is supertype of itself and subtype of A and B. The 1-influence of D contains D, G and H as D is their single closest 1-pole. The 1-influences are surrounded by a blob on Figure 1. □

Fact 1 *Let* $Pole^i_m = \{T^1_p, \ldots, T^l_p\}$. *Then* $\{Influence^i_m(T^1_p), \ldots, Influence^i_m(T^l_p)\}$ *is a partition of* $Dynamic^i_m$, *and for all* T *in* $Dynamic^i_m$, *we have:*

$$T \in Influence_m^i(T_p^k) \Leftrightarrow min_{\preceq}\{T_p \in Pole_m^i \mid T \preceq T_p\} = \{T_p^k\}$$

Pole of a Type. We define the function $pole_m^i$ to take a type T in $Dynamic_m^i$ and to return the i-pole type such that T is in its influence:

$$pole_m^i(T) = T_p, \text{ s.t. } T_p \in Pole_m^i \text{ and } T \in influence_m^i(T_p)$$

From this definition, and Fact 1, follows:

Corollary 1 $min_{\preceq}\{T_p \in Pole_m^i \mid T \preceq T_p\} = \{pole_m^i(T)\}$

4.2 Dispatch Trees

We define the dispatch tree (V, v_1) of m, where V is the set of vertices and v_1 is the root vertex. The tree is built using function $succ(v, T)$, that associates with a vertex v and a local pole T a unique successor v'. The set of local poles of v is noted $Pole^v$.

Vertices. The set of vertices of depth i, noted V_m^i, is defined as:

- $V_m^1 = \{v_1\}$.
- $\forall i \in \{2, \ldots, n\},\ V_m^i = \{succ(v_{i-1}, T^p) \mid v_{i-1} \in V_m^{i-1} \text{ and } T^p \in Pole_m^{v_{i-1}}\}$.

Applicable Methods. The methods *applicable on* a vertex v, noted $A(v)$, are defined as:

- $A(v_1) = M$.
- $\forall i \in \{2, \ldots, n\}, \forall v_i \in V_i,\ A(v_i) = \{m_k \mid v_i = succ(v_{i-1}, T^p), m_k \in A(v_{i-1}) \text{ and } T_k^{i-1} \succeq T^p\}$.

Local Poles, Local Dynamic Types. The local poles $Poles^v$ of a vertex v, the local dynamic types $Dynamic^v$ of v and the pole function $pole^v$ of v are defined as:

- $Pole^{v_1} = Pole_m^1$, $Dynamic^{v_1} = Dynamic_m^1$ and $\forall T \in Dynamic^{v_1}, pole^{v_1}(T) = pole_m^1(T)$.
- $\forall i \in \{2, \ldots, n\}, \forall v_i \in V_m^i$, let m' be the generic function which methods are those of $A(v_i)$, we have: $Pole^{v_i} = Pole_{m'}^i$, $Dynamic^{v_i} = Dynamic_{m'}^i$ and $\forall T \in Dynamic^{v_i}$, $pole^{v_i}(T) = pole_{m'}^i(T)$.

For each vertex v_n of V_m^n, there is a unique sequence $(v_1, T_1^p), \ldots, (v_{n-1}, T_{n-1}^p)$ such that $\forall i \in \{2, \ldots, n\},\ v_i = succ(v_{i-1}, T_{i-1}^p)$. Again, we note m' a generic function which methods are those of $A(v_n)$. For all T_n^p in $Pole_m^{v_n}$, we define $succ(v_n, T_n^p) = MSA(m'(T_1^p, \ldots, T_n^p))$.

The following theorem expresses that dispatch trees can give the MSA method of an invocation:

Theorem 1 *Let* $(T^1, \ldots, T^n) \in \Theta^n$. *If* $m(T^1, \ldots, T^n)$ *is well-typed and* $m_s = MSA(m(T_1, \ldots, T_n))$, *then the sequence* $(v_1, \ldots, v_n) \in V_m^1 \times \ldots \times V_m^n$ *is defined such that* $\forall i \in \{1, \ldots, n\}$, $T^i \in Dynamic^{v_i}$, *if* $i < n$, $v_{i+1} = succ(v_i, pole^{v_i}(T^i))$, *and* $m_s = succ(v_n, pole^{v_n}(T_n))$.

If $m(T^1, \ldots, T^n)$ *is not well-typed, then* $\exists i_0 \in \{1, \ldots, n\}$ *and a sequence* $(v_1, \ldots, v_{i_0}) \in V_m^1 \times \ldots \times V_m^{i_0}$ *such that* $\forall i \in \mathbb{N}^*,\ i < i_0,\ T^i \in Dynamic^{v_i}$ *and* $v_{i+1} = succ(v_i, pole^{v_i}(T^i))$, *and* $T^{i_0} \notin Dynamic^{v_{i_0}}$.

Proof: See Appendix A.

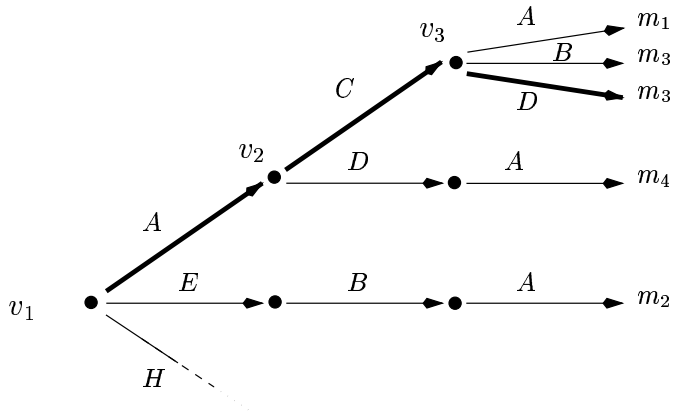


Figure 6: Path in a Dispatch Tree

Example 4.6: m being the generic function defined in Figure 4, Figure 6 shows the path taken in the dispatch tree for the invocation $m(D, F, H)$. We have $pole^{v_1}(D) = pole_m^1(D) = A$, which leads to vertex v_2. Then $pole^{v_2}(F) = C$, which leads to v_3. Finally $pole^{v_3}(H) = D$ and $succ(v_3, D) = m_3 = MSA(m(D, F, H))$. □

5 Size Optimizations

In this Section, we present two directions to optimize the size of dispatch trees. We describe them based on the basic dispatch tree presented above. However, they can be integrated in a global optimization algorithm.

We illustrate these optimizations on a more complex generic function obtained by adding an argument to the generic function given in Figure 4. This function, as well as its dispatch tree, appears in figure 7. There are 4 argument positions and 20 vertices.

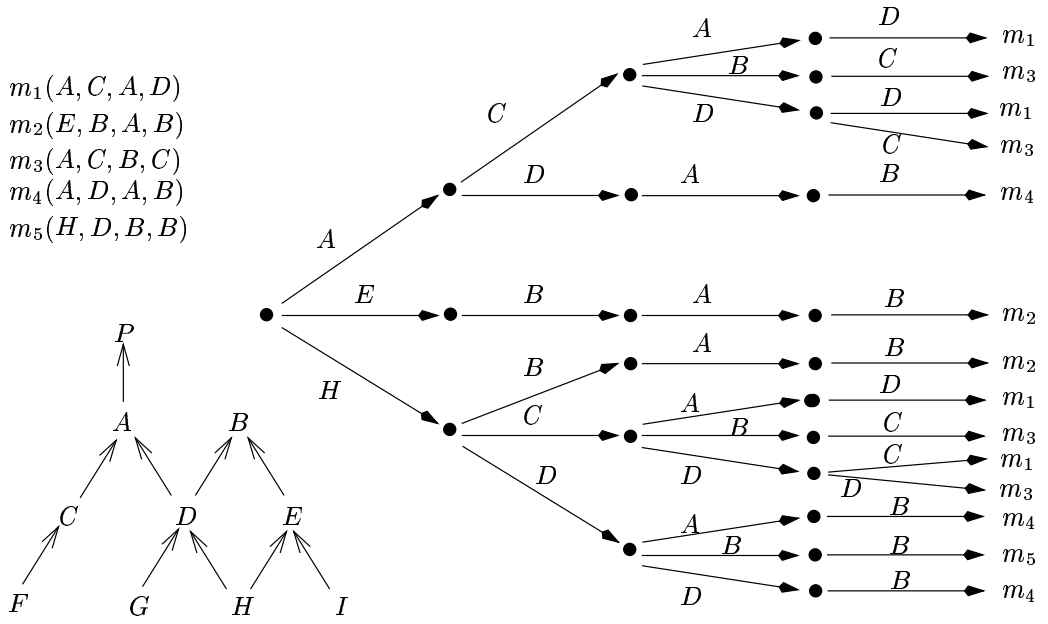


Figure 7: Dispatch Tree with 4 Arguments

5.1 Argument Position Ordering

The choice of considering the argument positions 1,2,...,n respectively in $V_m^1, \ldots, V_m^n$ is arbitrary. For example, the root vertex v_1 could be related to a position j, so that the successors of v_1 depend on the elements of $Pole_m^j$. Then at run-time, the first argument-

array would be used against the type of the jth argument of the invocation. This flexibility to choose the main position allows to shorten the dispatch tree.

We call *main position* at depth i the argument position that is considered at depth i. This position is chosen among the *active positions*, which are the argument positions not chosen as main position at smaller depths.

Example 5.1: Consider the type hierarchy and methods in Figure 7. If the positions are taken in the order 2,4,1,3 we obtain a smaller dispatch tree of 14 vertices, as shown in Figure 8. On the contrary, an order 3,1,4,2 yields a tree of 27 vertices. □

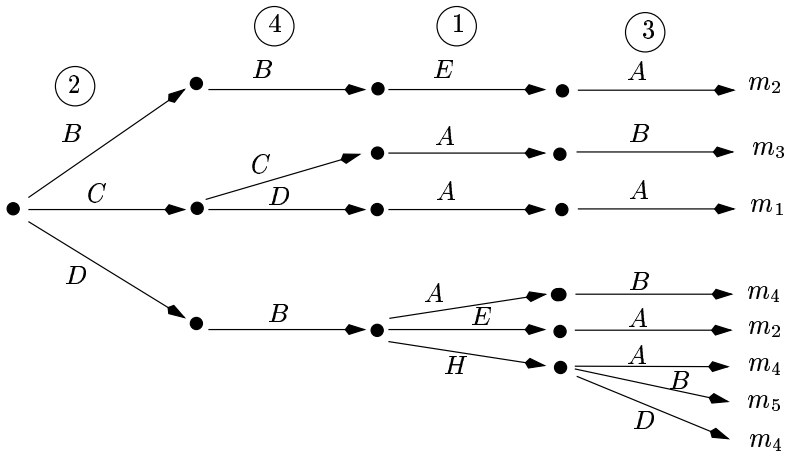


Figure 8: Uniform Argument Ordering

This ordering of positions is a *uniform ordering*, as the same order of positions applies to the whole tree. It is possible to use a *local ordering*, i.e. different main positions in different sub-trees. In this case, the main position, as well as the active positions, depend on the vertex considered and not on the depth. This greater flexibility allows to optimize the tree further. However dispatch time then takes some more time. Indeed, the order of main positions for an invocation depends on the path taken at run-time in the dispatch tree, and cannot be known at compile-time. It is necessary to store the main positions in each vertex to read them at run-time, which increases the time needed for dynamic dispatch. Moreover,

this storage of the main positions also needs a small amount of memory, e.g. one byte in each vertex.

Example 5.2: Consider the types and methods in Figure 9. The main position of v_1 is 2. In the subtree associated with C as second argument, the order of the main positions is 2,1,4,3. In the other subtrees this order is 2,4,1,3. The resulting tree has 13 vertices. Consider the run-time invocation $m(D, F, H, E)$. The first main position is statically known to be 2, and $pole^2_m(F) = D$, leading to v_2. It is necessary to read in v_2 the main position, here 4, to find that the fourth argument of the invocation, namely E, must be used to find the successor vertex v_3 with $pole^{v_2}(pole^4_m(E)) = pole^{v_2}(B) = B$. The main position at v_3 is 1, and $pole^{v_3}(pole^1_m(D)) = pole^{v_3}(A) = A$, which leads to v_4. The last main position is 3, and $pole^{v_4}(pole^3_m(H)) = pole^{v_4}(D) = B$, thus the MSA method is m_4. □

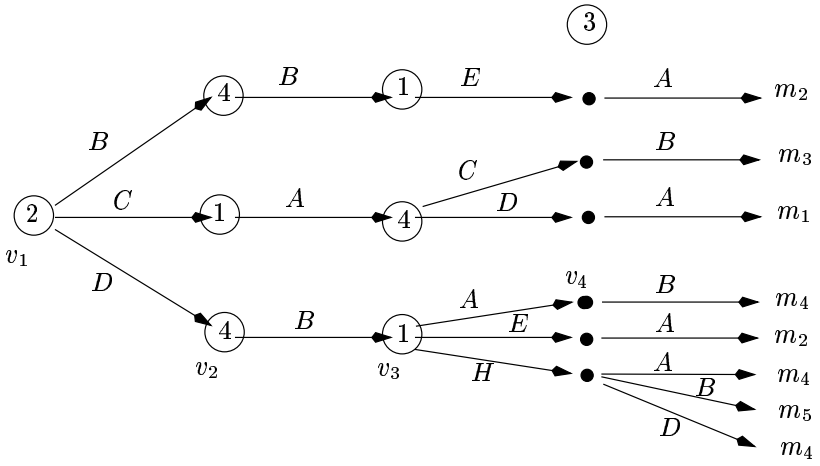


Figure 9: Variable Argument Ordering

Finding an efficient algorithm to compute the best position ordering is an open issue. The naïve approach is to compute the trees for each possible ordering, and then to compare the sizes. Instead, one may try to find this ordering incrementally, i.e. define the main position of vertices at depth $i + 1$ after having built the vertices at depth i. In general, it is not possible to predict the exact size of the following depths without building them. But intuitively, dispatch trees may compress better than dispatch tables because the successor

vertices are built using only the methods applicable to the current vertices. Indeed, Lemma 1 shows that fewer methods yield fewer local poles, i.e. fewer next vertices. Consequently the better reduction of the number of applicable methods is a good criteria for choosing the main position. It makes it necessary to compute the local poles in each active position for each vertex, together with the methods applicable to these poles, instead of doing this for only one position as in the approach with no ordering.

However, for the next to last depth, it is possible to determine the exact size of all remaining depths for each position as main position, because the number of global poles is known for each position, as well as the number of vertices at depth $n-1$.

5.2 Vertices Unification

It is sometimes possible to unify two vertices of a given depth, into one vertex that is the successor of two vertices. Two vertices can be unified iff their subtrees involve the same local poles and lead to the same method selections. Their unification spares the space taken by one vertex and all its subtree. When unifications occur, the dispatch structure becomes a direct, acyclic graph.

Example 5.3: Going back to Figure 7, we can see that two unifications can be done. The first unifies the vertices $succ(succ(v_1, A), C)$ and $succ(succ(v_1, H), C)$, and the second unifies $succ(succ(succ(v_1, A), D), A)$ and $succ(succ(succ(v_1, H), D), A)$. This spares 5 vertices, i.e. 15 memory words. □

Theorem 2 *If the only method precedence ordering is argument subtype precedence, with* $i \in \{1, \ldots, n\}$*, two elements* v_i *and* v'_i *of* V^i_m *can be unified if* $A(v_i) = A(v'_i)$.

Proof: See Appendix B.

Theorem 2 applies to the simple case of main positions taken in order. In the general case of a dispatch tree in which the main position depends on the vertex, the theorem should include as additional condition, the fact that the same set of main positions must be used in the paths from v_1 to v_i and from v_1 to v'_i.

6 Implementation and Cost Evaluation

6.1 Two-Level Mapping

In compressed dispatch tables, argument-arrays store the values of the functions $pole^i_m$. This storage is essential to allow dynamic dispatch to be done in constant time. However, storing the local pole of each type in each vertex would take up $|\Theta|$ memory words by vertex. We propose to use instead a two-level mapping. The first level uses *global* argument-arrays, associated with the whole tree, that map Θ to $Pole^i_m$ in each dimension i. The second level uses *local* argument-arrays, associated with each vertex, that map $Pole^i_m$ to the successors of the vertex. Theorem 3 expresses that this mapping allows to find $pole^{v_i}(T)$ at run-time. We first introduce the following lemma:

Lemma 1 *Let m and m' be two n-ary generic functions such that the set of methods of m' is included in the set of methods of m. Then, for all i in $\{1,\ldots,n\}$, we have $Pole^i_{m'} \subset Pole^i_m$ and for all type T in Θ, we have $pole^i_{m'}(T) = pole^i_{m'}(pole^i_m(T))$.*

Proof: See Appendix C.

Theorem 3 *For all $T \in \Theta$, all $i \in \{1,\ldots,n\}$ and all $v_i \in V^i_m$, we have $pole^{v_i}(T) = pole^{v_i}(pole^i_m(T))$.*

Proof: See Appendix D.

It is not necessary to store in each vertex v_i the maps from $Pole^i_m$ to $Pole^{v_i}$ and from $Pole^{v_i}$ to V^{i+1}_m. Instead, it suffices to store a map from $Pole^i_m$ to V^{i+1}_m. Indeed, for each type T_i in the invocation, $succ(v, pole^{v_i}(T))$ is the only value needed at run-time. Concerning the first argument position, as there is only one vertex v_1, it is not necessary to have a correspondence from Θ to $Pole^1_m$, then from $Pole^1_m$ to V_2. It suffices to have a direct correspondence from Θ to V_2. Hence we consider that the first global argument-array is a map from Θ to V^2_m, and v_1 is in fact not represented at run-time.

Consequently, the final structure is made of n global argument-arrays of size $|\Theta|$, and of local argument-arrays of size $|Pole^i_m|$ associated with each vertex v_i of $V_2,\ldots,V_n$.

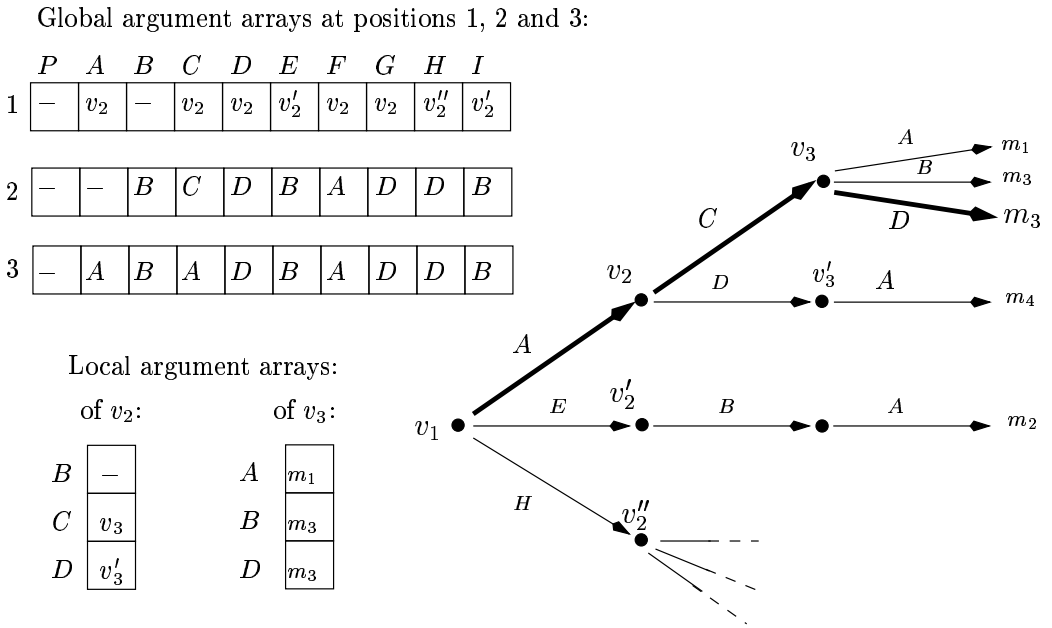


Figure 10: Argument-arrays in a Dispatch Tree

Example 6.1: Going back to the types and methods of Figure 4, the argument arrays associated with the vertices used in the case of invocation $m(D, F, H)$ are given in Figure 10. We can observe that the sizes of these arrays do not depend on the number of local poles, but only on the number of poles. The number of local poles only defines the number of successor vertices. □

6.2 Run-Time Dispatch

Given the global argument-arrays and the dispatch tree, computing at run-time the MSA method of an invocation $m(o_1, \ldots, o_n)$ is done as shown in Figure 11 in the case of a statically-typed language. For each argument position i, one first computes the global pole of the type of the run-time object o_i. This global pole allows then to find the successor vertex of v_i. As mentioned above, the first argument position is a special case. From the type of o_1, the first global argument-array directly leads to the vertex of V_2.

In a dynamically typed language the invocation should also include run-time type checks. This comes down to verify that the values of variables $v_2, \ldots, v_n$ and `msa` are not null before computing the next vertex or invoking `msa`.

$v_2 \leftarrow m\_arg\_1[o_1.type\_index]$
$v_3 \leftarrow v_2.local\_array[m\_arg_2[o_2.type\_index]]$
.....
$v_n \leftarrow v_{n-1}.local\_array[m\_arg_{n-1}[o_{n-1}.type\_index]]$
msa $\leftarrow v_n.local\_array[m\_arg_n[o_n.type\_index]]$
call msa$(o_1, \ldots, o_n)$

Figure 11: Dynamic Dispatch Using a Compressed Tree

Compared to the dynamic dispatch code using a dispatch table (see [AGS94]), we can see that each argument position, but the first, needs a suplementary dereferencing, in order to reach the following vertex using global poles. Dynamic dispatch is thus a little more expensive in time with dispatch trees.

6.3 Memory Cost

The memory cost of the dispatch tree includes the size $|\Theta|$ of each global argument-array (as for dispatch tables), and takes into account the substitution of the array of v_1 with the argument-array at position 1. Finally the size is:

$$\sigma_n = n \times |\Theta| + \sum_{i=2}^{n} (|V_m^i| \times |Pole_m^i|)$$

As function $succ(v, T_p)$ gives a unique result for each vertex v and each local pole T_p, for all $i \in \{2, \ldots, n\}$ we have $|V_m^i| = \sum_{v_{i-1} \in V_m^{i-1}} |Pole^{v_{i-1}}|$.

In the particular case of $n = 2$, we have:

$$\sigma_2 = 2 \times |\Theta| + |Pole_m^1| \times |Pole_m^2| \quad (1)$$

The latter size is exactly the size of the corresponding dispatch table. Let us assume now that $n > 2$. Let π_m^i be the average number of local poles per vertex at argument position i. We have:

$$|V_m^i| = \sum_{v_{i-1} \in V_m^{i-1}} |Pole^{v_{i-1}}| = |V_m^{i-1}| \times \pi_m^{i-1}$$

Thus the size of the dispatch tree is:

$$\sigma_n = n \times |\Theta| + |Pole_m^1| \times |Pole_m^2| + \sum_{i=3}^{n} (Pole_m^1 \times (\prod_{j=2}^{i-1} \pi_m^j) \times |Pole_m^i|) \qquad (2)$$

In comparison, the size of the dispatch table is:

$$\tau_n = n \times |\Theta| + \prod_{i=1}^{n} |Pole_m^i|$$

As n grows, the last product in (2) is prominent; if the average number of local poles in dimensions $2, \ldots, n-1$ is small enough, σ_n becomes smaller than τ_n.

6.4 Compressed Dispatch Tree Construction

The algorithm for dispatch tree construction is given in Figures 12 and 13. This algorithm simply yields the set of global argument-arrays, as the first of them refers to the vertices of V_2, which allow to reach the whole tree. The algorithm performs in two steps, the first of which being the computation of global poles. These are first used to build the global argument-arrays $m\_arg_1, \ldots, m\_arg_n$. At this point m_arg_1 holds global poles. The second step uses a temporary v_1 vertex. The vertices are implemented as tuple-structured objects, with the attributes *depth* (1 for v_1), *applicables* (M for v_1), and *local_array*. The creation of the local array of v_1 is based on the function *succ*, which recursively builds the entire tree. When the local argument-array of v_1 is created, it is combined with m_arg_1 to build the global argument-array at position 1, which holds the references to V_2, making v_1 useless.

The algorithm *succ* that builds a successor vertex appears in Figure 13. Building a vertex v involves computing (i) $A(v)$ as $v.applicable$ and $Pole^v$ using $A(v)$, (ii) the successor vertex for each member of $Pole^v$, and (iii) the local argument-array. At step (i), we know from Theorem 3 that $\{pole_m^{v_i}(T) \mid T \in \Theta\}$ equals to $\{pole_m^{v_i}(pole_m^i(T)) \mid T \in \Theta\}$, that itself equals to $\{pole_m^{v_i}(T_p) \mid T_p \in Pole_m^i\}$. Hence the computation of $Pole^v$ can be restricted

```
Input    : a generic function m, an ordered list of types Θ≤
Output   : a n-tuple of global argument-arrays (global_array_1, ..., global_array_n)

M ← methods(m)
Step 1: Creation of the global argument-arrays that hold global pole indices
for i in {1, ..., n} do
   Pole_m^i ←Poles(Θ≤, M, i)          // compute poles and influences
   for T in Θ≤ do                     // T.pole is a side-effect of pole computation
      m_arg_i[T.number] ←T.pole.number

Step 2: Creation of the global argument-array for dimension 1
v ← newVertex
v.depth ← 1
v.A ← M
for T^p in Pole_m^1 do
   v.local_array[T^p.number] ← succ(v, T^p)

for T in Θ≤ do
   m_arg_1[T.number] ← v.local_array[m_arg_1[T.number]]

return((m_arg_1, ..., m_arg_n))
```

Figure 12: Dispatch Tree Creation Algorithm

to the global poles. This speeds up the computation and directly gives the mapping from $Pole_m^i$ to $Pole^v$, that is needed in the local arrays. Step (ii) simply consists in the recursive invocation of *succ* and step (iii) is a join of the relationships (global pole,local pole) and (local pole,successor vertex).

The computation of $Pole^v$ is done using the pole computation algorithm, which description appears in a forthcoming report [ADS96]. This algorithm yields the set of poles and as a side-effect, relates each type T in the input set with its pole as its attribute $T.pole$. This algorithm needs that the input types are given in an order that is a linear extension of the subtyping relation, and that each type refers to its direct supertypes. Here the input types are the global poles, which are produced by a previous invocation of *Poles*, hence they are already ordered. Moreover, the supertype relationship between poles is also computed by this previous invocation, as the set of closest supertype poles of each pole.

Input : a vertex v, a local pole T^p_{pred}
Output : a new vertex v'

Step 1 : Local Poles Computation
$v' \leftarrow$new Vertex
$i \leftarrow v.depth + 1$
$v'.applicable \leftarrow \{m_k \in v.applicable \mid T^k_{i-1} \succeq T^p_{pred}\}$
$Pole^v \leftarrow Poles(Pole^i_m, v'.applicable, i)$

Step 2 : Successor Vertices Construction
for T^p in $Pole^v$ do
  $T^p.succ \leftarrow succ(v', T^p)$

Step 3 : Successor Vertices Storage in the Local Array
for T in $Pole^i_m$ do // $T.pole$ is a side-effect of pole computation
  $v'.local\_array[T.number] \leftarrow T.pole.succ$

return(v')

Figure 13: *succ* Algorithm

6.5 Experimental Results

Our results have been obtained using the Cecil compiler as application. As it is written in Cecil itself, this compiler is a real application with multi-methods. It includes 932 types, and 3990 generic functions.

Many of these functions have 0 or 1 target argument, i.e. an argument which is really used at run-time for method dispatch. Indeed, despite many generic functions have many arguments, it often occurs that their type is the same for all methods of the generic function. For our tests, we consider an argument is a target only if there are at least two global poles defined for its position. Based on (1), we do not consider generic functions with two target arguments as their compressed dispatch tables need as much memory as compressed trees. Hence we only consider generic functions with 3 and 4 targets.

As the argument-arrays have a similar contents as in the case of dispatch tables, we only compare the sizes of the trees and tables themselves. We assume that dispatch tables contain method code addresses, while dispatch trees are made of local argument-arrays that contain either next vertices' addresses, or method code addresses (in leaf vertices). As both contain only addresses, we define a *cell* of a dispatch table or dispatch tree as a memory

block which length is the system's address length. We can then compare dispatch tables and trees, based on their number of cells.

The results are given as an histogram in Figure 14, which compares dispatch tables with the dispatch trees obtained using all combinations of the optimizations presented in Section 5. The x-axis is indexed by the kinds of optimizations used, and the y coordinate gives the corresponding size, according to an arbitrary scale giving size 100 to dispatch tables.

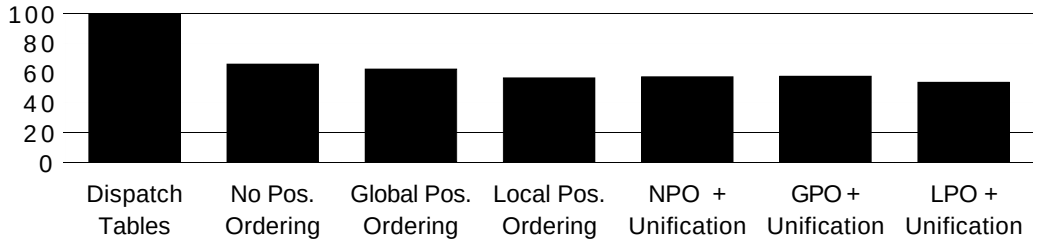


Figure 14: Size of Dispatch Structures

6.6 Optimized Cell Size

The main idea of this optimization is to represent data on small memory words, i.e. bytes or 16-bits words. Most modern processors can easily handle such words. This is possible here in the following cases:

1. When the number of global poles is less than 256, if the argument position is not the active position of v_1, the corresponding global argument-arrays can use bytes. Usually this number is less than 65536, hence at worse 16-bits words will be used.

2. Assume all vertices of a tree are stored contiguously, starting from a *base* address. Then the first argument-array may hold, instead of the addresses of the vertices of V_2, the offsets of these addresses w.r.t. the base address. Usually, all these offsets will be smaller than 2^{16}, which allows to store them on 16-bits or 8-bits memory words. This optimization makes it necessary, at run-time, to add the base address to the offset found in the first argument-array.

3. In the same way, the local argument-arrays may hold the offsets of the addresses of the next vertices w.r.t. the base addresse. Alternatively, this offset may be taken w.r.t. the base address of the local argument-array. This array should be closer to the next vertices, increasing the chances that all offsets be smaller than 256.

Focusing only on the representation of the tree itself (i.e., not on the argument-arrays), we apply the last optimization to the compressed dispatch trees of the Cecil compiler. We also consider the optimizations described above, except local conversion arrays. We assume a system where memory addresses are stored on 32-bits words.The results appear in Figure 15.

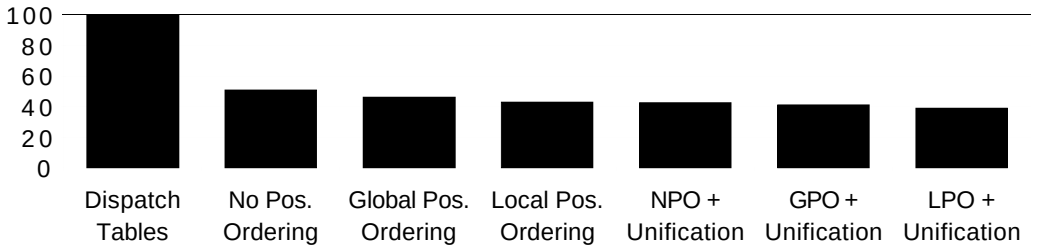


Figure 15: Sizes of Dispatch Structures with Optimized Cell Size

If we do not consider local position ordering, which slows down the dynamic dispatch, the better compression rate w.r.t. dispatch tables is obtained with global position ordering, vertex unification and cell size optimization. Dispatch trees then take 41% the size of a dispatch table.

7 Conclusion and Related Work

In this report, we have shown that dispatch trees can be compressed, allowing constant-time run-time dispatch of multi-methods using little memory. Dispatch trees are more complex to build than dispatch tables, because they involve more pole computations. This is especially true if one tries to optimize the compression of dispatch trees. Run-time dispatch is also a little longer with dispatch trees, because it involves a successive dereferencing of all vertices that build the path leading to the MSA method. However compressed dispatch trees need

less memory than compressed dispatch tables, because the subtree of each vertex is defined for a smaller number of applicable methods than the whole dispatch tree.

The approach of [CTK94] is similar to ours in that it also relies on a tree (called a dispatch automaton). It supports languages which precedence ordering is Inheritence Order Precedence, as defined in [ADL91]. This order is a particular case of Argument Subtype Precedence with monotonicity, that we assume in this report. No language currently supports the former order[1], while both Cecil and Dylan support the latter. Their approach offers the dynamic dispatch of multi-methods in bounded time. It uses a dispatch tree without global poles, hence there are no global argument-arrays. Each vertex contains the list of its local poles, and the list of corresponding references to the successor vertices. The correspondance between run-time types and local poles must be computed at run-time. This computation is done in time proportional to the number of poles. To garantee that selection time is bounded by a constant, the vertices have a different structure when the number of their local poles is greater than an arbitrary constant, the same for all vertices. Their structure is then simply an array that associates with each possible type the successor vertex.

The tree construction algorithm in [CTK94] takes advantage of the ordering of methods due to Inheritance Order Precedence, by incrementally sorting applicable methods. In the leaf vertices, MSA methods appear as first element of applicable method lists, avoiding method comparisons. As a consequence, there is no possible position ordering, as the order must be that of the arguments, because Inheritance Order Precedence depends on this order. They propose the vertice unification optimization, and [CT95] also gives an optimization of the search of vertices with which a given vertex can be unified. Again, this only applies to languages supporting Inheritance Order Precedence.

Acknowledgment: We would like to thank Jeff Dean who has given us the relevant information about the types and methods of the Cecil compiler.

[1] Inheritance Order Precedence was presented in [ADL91] to model the precedence algorithm of CLOS, however the former is obviously monotonic, while the latter is not, as shown in [DHHM92]

8 Appendices

A Proof of Theorem 1

We proove the theorem by induction on the argument position i, and we begin with the first part of the theorem. Assuming that $m(T^1,\ldots,T^n)$ is well-typed and $m_s = MSA(m(T_1, \ldots,T_n))$, we first show that $\forall i \in \{1,\ldots,n\}$, $m_s \in A(v_i)$ and $T^i \in Dynamic^{v_i}$.

- As $A(v_1) = M$, $m_s \in A(v_1)$. As m_s is applicable to $(T^1,\ldots,T^n)$, $T_1 \in Dynamics^{v_1}$.
- Let us assume $m_s \in A(v_{i-1})$ and $T^{i-1} \in Dynamic^{v_{i-1}}$. v_i is defined as $v_i = succ(v_{i-1}, pole^{v_{i-1}}(T^{i-1}))$. Let m' be the generic function whose methods are the methods of $A(v_{i-1})$. As T_s^{i-1} is a primary $i-1$-pole of m', from Corollary 1 we have $T_s^{i-1} \succeq pole_{m'}^{i-1}(T^{i-1})$, i.e. $T_s^{i-1} \succeq pole^{v_{i-1}}(T^{i-1})$, hence $m_s \in A(v_i)$. As m_s is applicable to $(T^1,\ldots,T^n)$, $T_i \in Dynamics^{v_i}$.

Let us show now that $m_s = succ(v_n, pole^{v_n}(T_n))$. For all $i \in \{1,\ldots,n\}$, we note $T_p^i = pole^{v_i}(T^i)$. As $T_s^n \in Pole^{v_n}$ and $T_s^n \succeq T^n$, we have $T_s^n \succeq T_p^n$. Hence m_s is applicable to $(T_p^1,\ldots,T_p^n)$, and $(T_p^1,\ldots,T_p^n) \succeq (T^1,\ldots,T^n)$. By monotonicity, if m_s is not the MSA method of $(T_p^1,\ldots,T_p^n)$, it is not the MSA method of $(T^1,\ldots,T^n)$ either, which is false by definition of m_s. Hence $m_s = MSA(T_p^1,\ldots,T_p^n) = succ(v_n, T_p^n)$. This concludes the first part of this proof.

Let us now assume that $m(T^1,\ldots,T^n)$ is not well-typed. If $\forall i \in \{1,\ldots,n\}$ $\exists m_k$ s.t. $(T_k^1,\ldots,T_k^i) \succeq (T^1,\ldots,T^i)$, $m(T^1,\ldots,T^n)$ would be well-typed, a contradiction. If $\nexists m_k$ s.t. $T_k^1 \succeq T^1$, the theorem is trivially true with $i_0 = 1$. If not, let $i_0 = min\{i \mid \nexists\ m_k$ s.t. $(T^1,k,\ldots,T_k^i) \succeq (T^1,\ldots,T^i)\}$. We have $i_0 \geq 2$. Let also m_l s.t. $\forall i < i_0$, $T_l^i \succeq T^i$. We first show that $\forall i < i_0$, $m_l \in A(v_i)$ and $T^i \in Dynamic^{v_i}$.

- As $A(v_1) = M$, $m_l \in A(v_1)$. As m_l is applicable to $(T^1,\ldots,T^n)$, $T_1 \in Dynamics^{v_1}$.
- For $1 < i < i_0$, let us assume $m_l \in A(v_{i-1})$ and $T^{i-1} \in Dynamic^{v_{i-1}}$. v_i is defined as $v_i = succ(v_{i-1}, pole^{v_{i-1}}(T^{i-1}))$. Let m' be the generic function whose methods are the methods of $A(v_{i-1})$. As T_l^{i-1} is a primary $i-1$-pole of m', from Corollary 1 we have

$T_l^{i-1} \succeq pole_{m'}^{i-1}(T^{i-1})$, i.e. $T_l^{i-1} \succeq pole^{v_{i-1}}(T^{i-1})$, hence $m_l \in A(v_i)$. As $T_l^i \succeq T^i$, $T_i \in Dynamics^{v_i}$.

Finally, let $v_{i_0} = succ(v_{i_0-1}, pole^{v_{i_0-1}}(T^{i_0-1}))$. If $T^{i_0} \in Dynamics^{v_{i_0}}$, $\exists m_k \in A(v_{i_0})$ s.t. $T_k^{i_0} \preceq T^{i_0}$. Moreover $\forall i < i_0$, $m_k \in A(v_i)$, hence $1 < i < i_0 \Rightarrow T_k^i \succeq pole^{v_i}(T^i)$, consequently $(T_k^1, \ldots, T_k^{i_0}) \preceq (T^1, \ldots, T^{i_0})$. This is impossible by construction of i_0, hence $T^{i_0} \notin Dynamics^{v_{i_0}}$. This concludes our proof.

B Proof of Theorem 2

Let $T^1, \ldots, T^{i-1}$ and $T^{1\prime}, \ldots, T^{i-1\prime}$ s.t. $v_i = succ(\ldots(v_1, T^1), \ldots, T^{i-1})$ and $v_i' = succ(\ldots(v_1, T^{1\prime}), \ldots, T^{i-1\prime})$.

We first assume that $i < n$. Let $T^i \in Pole^{v_i}$. As $Pole^{v_i'}$ derives from the primary poles, that are given by the signatures of $A(v_i')$, we have $Pole^{v_i} = Pole^{v_i'}$, hence $succ(v_i', T^i)$ is defined.

Moreover, we have $A(succ(v_i, T^i)) = A(succ(v_i', T^i))$. Indeed:

$m_k(T_k^1, \ldots, T_k^n) \in A(succ(v_i, T^i)) \Leftrightarrow m_k \in A(v_i)$ and $T_k^i \succeq T^i \Leftrightarrow m_k \in A(succ(v_i', T^i))$.

By induction, $\forall v_n = succ(\ldots(v_i, T^i), \ldots, T^{n-1})$, $v_n' = succ(\ldots(v_i', T^i), \ldots, T^{n-1})$ is defined and $A(v_n) = A(v_n')$. This is also trivially true if $i = n$. We now assume $i \leq n$.

As above, $\forall T^n \in Pole^{v_n}$, $succ(v_n', T^n)$ is defined. Let us show that $succ(v_n, T^n) = succ(v_n', T^n)$, i.e. that $MSA(m(T^1, \ldots, T^n)) = MSA(m(T^{1\prime}, \ldots, T^{i-1\prime}, T^i, \ldots, T^n))$.

We note $m_k(T_k^1, \ldots, T_k^n) = MSA(m(T^1, \ldots, T^n))$ and $m_k' = MSA(m(T^{1\prime}, \ldots, T^{i-1\prime}, T^i, \ldots, T^n))$. As $m_k \in A(v_n)$, $m_k \in A(v_n')$, and $T_k^n \succeq T^n$, hence m_k is applicable to $(T^{1\prime}, \ldots, T^{i-1\prime}, T^i, \ldots, T^n))$. If argument subtype precedence is the only method ordering, it means that $m_k \succeq m_k'$. In the same way, $m_k' \succeq m_k$. Thus $m_k = m_k'$, i.e. v_n' has the same successors as v_n. This means that the subtree having v_i as root is included in the subtree having v_i' as root; the converse being obviously true, v_i and v_i' can be grouped. This concludes our proof.

C Proof of Lemma 1

Let $i \in \{1, \ldots, n\}$. We assume an ordering (T_k^p) of $Pole_{m'}^i$ such that $T_k^p \succ T_{k'}^p \Rightarrow k < k'$. We prove Lemma 1 by induction on k.

We first prove that $T_1^p \in Pole_m^i$. If T_1^p was not a primary pole, there would exist T_a^p and T_b^p in $Pole_m^i$ such that $T_a^p, T_b^p \succeq T_1^p$, hence $a < 1$, which is impossible. As T_1^p is a primary pole, it is in the signature of some method of m', hence of m, i.e. $T \in Pole_m^i$.

We now assume that the lemma is true up to T_{k-1}^p. If T_k^p is a primary pole, then $T_k^p \in Pole_m^i$ as above. If not, $|closest - poles_{m'}^i(T_k^p)| > 1$. Hence let $T_a^p, T_b^p \in closest\text{-}poles_{m'}^i(T_k^p)$, s.t. $T_a^p \neq T_b^p$.

As $T_a^p, T_b^p \succ T_k^p$, we have $a, b < k$ hence $T_a^p, T_b^p \in Pole_m^i$. We consider the set $E = closest\text{-}poles_m^i(T_k^p)$. We prove that $|E| > 1$, which implies that $T_k^p \in Pole_m^i$. First, $E \neq \emptyset$ because $T_a^p \in \{T \in Pole_m^i \mid T \succ T_k^p\}$.

Let us assume that $E = \{T_c^p\}$. We have then $T_c^p \prec T_a^p$ and $T_c^p \prec T_b^p$. If $closest\text{-}poles_{m'}^i(T_c^p) = \{T_d^p\}$, we have $T_d^p \prec T_a^p$. As $T_d^p \succ T_c^p \succ T_k^p$ and $T_d^p \in Pole_{m'}^i$, this implies that $T_a^p \notin closest\text{-}poles_{m'}^i(T_k^p)$, in contradiction with T_a^p's construction. As $T_c^p \prec T_a^p$, $|closest\text{-}poles_{m'}^i(T_c^p)| > 1$, hence $T_c^p \in Pole_{m'}^i$. Again, this is impossible by construction of T_a^p.

Hence $|E| > 1$, which means that $T_k^p \in Pole_m^i$ and concludes our proof of the lemma.

D Proof of Theorem 3

We first remark that for any generic function m', $T_a \preceq T_b \Rightarrow pole_{m'}^i(T_a) \preceq pole_{m'}^i(T_b)$. Indeed, we have $pole_{m'}^i(T_b) \succeq T_b \succeq T_a$, and $pole_{m'}^i(T_b) \in Pole_{m'}^i$. As $pole_{m'}^i(A) = min_{\preceq}\{T_p \in Pole_{m'}^i \mid T_p \succeq T_a\}$, we have $pole_{m'}^i(A) \preceq pole_{m'}^i(T_b)$.

Let m' be the generic function which methods are in $A(v_i)$, $T_v = pole^{v_i}(T) = pole_{m'}^i(T)$, $T_p = pole_m^i(T)$ and $T_v' = pole^{v_i}(T_p) = pole_{m'}^i(T_p)$. We show that $T_v \preceq T_v'$, then that $T_v \succeq T_v'$, to conclude that $T_v = T_v'$.

As $T \preceq T_p$, we have $pole_{m'}^i(T_p) \preceq pole_{m'}^i(T_p)$, i.e. $T_v \preceq T_v'$.

From Lemma 1, $Pole^i_{m'} \subset Pole^i_m$, thus $T_v \in Pole^i_m$. As $T_p = min_{\preceq}\{T'_p \in Pole^i_m \mid T'_p \succeq T\}$, we have $T_p \preceq T_v$. Moreover, $T'_v = pole^i_{m'}(T_p)$ and $T_v = pole^i_{m'}(T_v)$ (i.e., T_v is its own i-pole in v_i). Consequently, $T'_v \preceq T_v$. This concludes our proof.

References

[ADL91] Rakesh Agrawal, Linda G. DeMichiel, and Bruce G. Lindsay. Static type checking of multi-methods. In *Proc. OOPSLA*, 1991.

[ADS96] E. Amiel, E. Dujardin, and E. Simon. Optimizing multi-methods dispatch using compressed dispatch tables. Technical report, INRIA, 1996. To appear.

[AGS94] E. Amiel, O. Gruber, and E. Simon. Optimizing multi-methods dispatch using compressed dispatch tables. In *Proc. OOPSLA*, 1994.

[App94] Apple Computer. *Dylan Interim Reference Manual*, June 1994. Available by ftp from ftp.cambridge.apple.com in /pub/dylan/dylan-manual.

[BDG+88] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya Keene, Gregor Kiczales, and David A. Moon. Common Lisp Object System specification. *SIGPLAN Notices*, 23, Sept. 1988.

[BKK+86] D. G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel. CommonLoops: Merging Lisp and object-oriented programming. In *Proceedings of the first International Conference on Object-Oriented Programming : Systems, Languages and Applications*, Portland, OR, September 1986.

[Cha92] Craig Chambers. Object-oriented multi-methods in Cecil. In *Proc. ECOOP*, 1992.

[CT95] W. Chen and V. Turau. Multiple dispatching based on automata. *TAPOS*, 1(1), 1995.

[CTK94] W. Chen, V. Turau, and W. Klas. Efficient dynamic look-up strategy for multi-methods. In *Proc. ECOOP*, 1994.

[CU91] Craig Chambers and David Ungar. Making pure object-oriented languages practical. In *Proc. OOPSLA*, 1991.

[DH95] K. Driesen and U. Hölzle. Minimizing row displacement dispatch tables. In *Proc. OOPSLA*, 1995.

[DHHM92] R. Ducournau, M. Habib, M. Huchard, and M.L. Mugnier. Monotonic conflict resolution mechanisms for inheritance. In *Proc. OOPSLA*, 1992.

[DHHM94] R. Ducournau, M. Habib, M. Huchard, and M.L. Mugnier. Proposal for a monotonic multiple inheritance linearization. In *Proc. OOPSLA*, 1994.

[DMPM89] R. Dixon, T. McKee, P.Schweizer, and M.Vaughan. A fast method dispatcher for compiled languages with multiple inheritance. In *Proc. OOPSLA*, pages 211–214, 1989.

[DS84] L. Peter Deutsch and Alan Schifman. Efficient implementation of the Smalltalk-80 system. In *Proc. ACM POPL*, 1984.

[ES92] M. A. Ellis and B. Stroustrup. *The annotated C++ reference manual.* Addison-Wesley, Reading, Mass., 1992.

[GR89] Adele Goldberg and David Robson. *Smalltalk-80, the language.* Addison-Wesley, 1989.

[HC92] S.-K. Huang and D.-J. Chen. Two-way coloring approaches for method dispatching in object-oriented programming system. In *Proceedings of the Annual International Computer Software and Applications Conference*, pages 39–44, Chicago, IL, 1992.

[KR90] Gregor Kiczales and Luis Rodriguez. Efficient method dispatch in PCL. In *Proc. ACM POPL*, 1990.

[Mel94] Jim Melton, editor. *(ISO Working Draft) SQL Persistent Stored Modules (SQL/PSM).* ANSI X3H2-94-331, August 1994.

[Mey92] Bertrand Meyer. *Eiffel, the language.* Prentice-Hall, 1992.

[MHH91] Warwick B. Mugridge, John Hamer, and John G. Hosking. Multi-methods in a statically-typed programming language. In *Proc. ECOOP*, 1991.

[SCB+86] C. Schaffert, T. Cooper, B. Bullis, M. Kilian, and C. Wilpot. An introduction to Trellis/Owl. In *Proc. OOPSLA*, 1986.

[UP83] David Ungar and David Patterson. *Smalltalk-80: Bits of History and Words of Advice*, chapter Berkeley Smalltalk: Who Knows Where the Time Goes? Addison Wesley, 1983.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 46 avenue F´elix Viallet, 38031 GRENOBLE Cedex 1
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399